CODE PATH *ORG

Functions and Modules

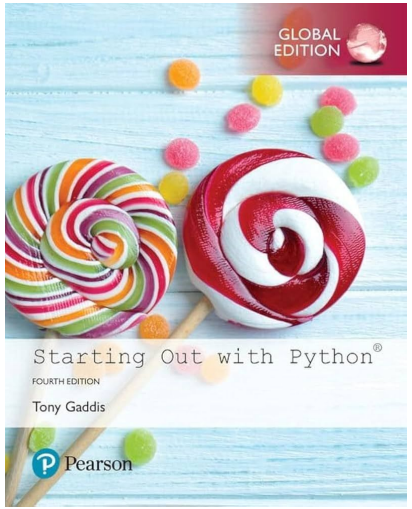
CIIC 3015

Karelys López Rivera

Lecture Goals

- **Functions**
  - What are they?
  - How are they used?
  - Live Coding

- **Modules**
  - What are they?
  - How to create and use them?
  - Examples
  - Live Coding



Starting Out With Python (4th Edition) Chapter 5

Recap

The and operator

The and operator returns `True` only if both conditions are `True`.

Condition 1	Condition 2	Result
`True`	`True`	`True`
`True`	`False`	`False`
`False`	`True`	`False`
`False`	`False`	`False`

The `or` operator

The `or` operator returns `True` if at least one condition is `True`.

Condition 1	Condition 2	Result
`True`	`True`	`True`
`True`	`False`	`True`
`False`	`True`	`True`
`False`	`False`	`False`

The `not` operator

The `not` operator reverses the value of a condition.

Condition	**Result**
`True`	`False`
`False`	`True`

Conditional statements

`if` statement:	`if-else` statement:	`if-elif-else` statement:

`if condition:`
`    execute code`
	`if condition:`
	`    execute code`
	`else:`
	`    execute code`
		`if condition:`
		`    execute code`
		`elif condition:`
		`    execute code`
		`else:`
		`    execute code`

IT'S CODING TIME

- Debugging errors

Scenario

Scenario:

- So far, we have created all code within a single giant block of code.
- For example, we used arithmetic operators to convert celsius to fahrenheit as follows:

```
# Converting Celsius to Fahrenheit
celsius = 20
fahrenheit = 9 / 5 * celsius + 32
```

Scenario:

- However, what if we need to convert three temperatures from celsius to fahrenheit?
- An option could be to do:

```
# Converting Celsius to Fahrenheit
celsius_1 = 20
fahrenheit_1 = 9 / 5 * celsius_1 + 32

celsius_2 = 30
fahrenheit_2 = 9 / 5 * celsius_2 + 32

celsius_3 = 40
fahrenheit_3 = 9 / 5 * celsius_3 + 32
```

Scenario:

- Are there any problems with this approach?

- Yes! It is **not scalable**
- For example, what if I ask you to convert the temperature of all 78 towns from celsius to fahrenheit and calculate the average temperature across them?

```
# Converting Celsius to Fahrenheit

celsius_1 = 20

fahrenheit_1 = 9 / 5 * celsius_1 + 32


celsius_2 = 30

fahrenheit_2 = 9 / 5 * celsius_2 + 32


celsius_3 = 40

fahrenheit_3 = 9 / 5 * celsius_3 + 32

                        ...

celsius_78 = 40

fahrenheit_78 = 9 / 5 * celsius_78 + 32
```

material made by: Jean Mendez

Scenario:

- Are there any more problems with this approach?
- What if there is a bug in the code? Now you have to fix it in many different places too!

```
# Converting Celsius to Fahrenheit

celsius_1 = 20

fahrenheit_1 = 9 / 5 * celsius_1 + 32


celsius_2 = 30

fahrenheit_2 = 9 / 5 * celsius_2 + 32


celsius_3 = 40

fahrenheit_3 = 9 / 5 * celsius_3 + 32

                        ...

celsius_78 = 40

fahrenheit_78 = 9 / 5 * celsius_78 + 32
```

material made by: Jean Mendez

Scenario:

- We can use **functions** to reuse a block of code.
- For example, we can create a function to reuse the logic of converting the temperature:

```
# Converts Celsius to Fahrenheit
def convert_celsius_to_fahrenheit(celsius):
    fahrenheit = 9 / 5 * celsius + 32
    return fahrenheit
```

Scenario:

Once we have created our function, we can use it as follows:

```
# Converts Celsius to Fahrenheit
def convert_celsius_to_fahrenheit(celsius):
    fahrenheit = 9 / 5 * celsius + 32
    return fahrenheit

# Use the function to quickly convert temperatures
fahrenheit_1 = convert_celsius_to_fahrenheit(20)
fahrenheit_2 = convert_celsius_to_fahrenheit(30)
fahrenheit_3 = convert_celsius_to_fahrenheit(40)
```

Functions

Functions

A **function** is a reusable block of code that performs a specific task such as manipulating data, making calculations, or processing text.

The benefits of using functions include:

- Reusability: Functions allow you to write a block of code once and use it multiple times, saving time and effort.
- Readability: Functions break down complex code into smaller, more manageable chunks, making your code easier to understand and maintain.
- Reduced errors: Using functions helps you avoid introducing errors by duplicating code.

Syntax: Define a function

To define a function:

- Start with the keyword `def`
- Then create a name for the function followed by parentheses and a colon
- Enclose any parameters inside the parentheses
- Indent the body of the function
- In the function body, add the code that will execute when the function is called

```
def greet(name):                # Definition of greet() function
    print(f"Hello, {name}!")
```

Examine the code

What keyword should be used in the highlighted section to define the `add_numbers` function?

```
??? add_numbers(num1, num2):
    print(f"{num1 + num2}")

add_numbers(11, 34)
```

Examine the code

What keyword should be used in the highlighted section to define the `add_numbers` function?

```
def add_numbers(num1, num2):
    print(f"{num1 + num2}")

add_numbers(11, 34)
```

- The keyword `def` should be used to define the `add_numbers` function.

Call functions

Calling a function means to ask the interpreter to run the code contained within the function's body.

- You call a function by using its name followed by parentheses.
- You must call a function after the function is defined (not before). Otherwise, you'll get an error message.

```
def greet(name):
    print(f"Hello, {name}!")

greet("Sasha")   # Calling the function
```

```
Hello, Sasha!
```

Function name guidelines

Functions names should be descriptive and indicate the purpose of the function. In addition, Python names should:

- Use lowercase with underscores to separate words (common convention in Python)
- Avoid special characters (use only letters, numbers, and underscores)
- Start with a letter
- Avoid reserved keywords

Examples of good function names

- `calculate_area()`
- `get_user_input()`
- `is_valid_email()`

Examples of bad function names

- `x()` (unclear what the function does)
- `user#input()` (uses special character)
- `1calculate_area()` (starts with a number, will result in syntax error)

Built-in functions

Python comes with many **built-in** functions that you can use right away.

- Built-in functions do not require a function definition in the code.
- As their name implies, they are built into the Python programming language to handle common tasks
- You are already familiar with some built-in functions, including:
  - `print()`: Displays a specified message to the screen
  - `input()`: Requests data from the user
  - `int()`: Casts a value to an integer
  - `str()`: Casts a value to a string

Custom vs. built-in functions

Feature	Custom functions	Built-in functions
Definition	User-defined blocks of code	Predefined functions in Python
Availability	Need to be defined first	Available from the start
Flexibility	More flexible - tailored to needs	Fixed purpose
Reusability	Reusable within your program	Reusable in any Python program
Control	More control over behavior	Limited control, specific tasks
Examples	`greet(name)`, `calculate_area(length, width)`	`print()`, `input()`
Suitable for	Specific tasks in your program	Common tasks, quick solutions

Check your understanding

What code should be used in the highlighted section to call the function?

```
def even_or_odd(number):
    if number % 2 == 0:
        print(f"{number} is even.")
    else:
        print(f"{number} is odd.")

???(10)
```

Check your understanding

What code should be used in the highlighted section to call the function?

```
def even_or_odd(number):
    if number % 2 == 0:
        print(f"{number} is even.")
    else:
        print(f"{number} is odd.")

even_or_odd(10)
```

- The function name `even_or_odd`

Parameters and arguments

Parameters and arguments

A **parameter** is a variable defined in the header of a function definition.

- Acts as a placeholder for a value that will be passed into the function when it is called
- Listed inside the parentheses following the function name

An **argument** is a value supplied to a function when it is called.

- Passed into the function's parameters
- Listed inside the parentheses when you call the function

Parameter

```
def greet(name):
    print(f"Hello, {name}!")
```

Argument

```
greet("Rosario")
```

Examine the code

What are the parameters of the `greet()` function?

```
def greet(first_name, last_name_initial):
    print(f"Hello, {first_name} {last_name_initial}.")

greet("Ariel", "Z")
```

Examine the code

What are the parameters of the `greet()` function?

```
def greet(first_name, last_name_initial):
    print(f"Hello, {first_name} {last_name_initial}.")

greet("Ariel", "Z")
```

- The `greet()` function has two parameters: `first_name` and `last_name_initial`

Example: Function with multiple parameters/arguments

```
# Function defined with two parameters: width and length
def calculate_area(width, length):
    area = width * length
    return area

# Function calls with different arguments
rectangle_1_area = calculate_area(5, 10)
rectangle_2_area = calculate_area(7, 12)
rectangle_3_area = calculate_area(3, 8)
```

Check your understanding

What arguments are passed into the `print_info()` function?

```
def print_info(name, age, city):
    print(f"Name: {name}")
    print(f"Age: {age}")
    print(f"City: {city}")

print_info("Kai", 22, "Tokyo")
```

Check your understanding

What arguments are passed into the `print_info()` function?

```
def print_info(name, age, city):
    print(f"Name: {name}")
    print(f"Age: {age}")
    print(f"City: {city}")

print_info("Kai", 22, "Tokyo")
```

- Three arguments are passed into the `print_info()` function: `"Kai"`, 22, and `"Tokyo"`

Return values

Return statements

A **return statement** is a line of code that returns a value back from a function to the location in the program where the function was called.

- Functions can return a value using the `return` keyword.
- This value can be stored in a variable and used later in your program
- The syntax of a `return` statement is the keyword `return` followed by the value that you want to return from the function.

Examine the code

What keyword should be used in the highlighted section to return the value of `total_cost`?

```
def calculate_order_total(price, quantity):
    total_cost = price * quantity
    ??? total_cost

calculate_order_total(10, 5)
```

Examine the code

What keyword should be used in the highlighted section to return the value of `total_cost`?

```
def calculate_order_total(price, quantity):
    total_cost = price * quantity
    return total_cost

calculate_order_total(10, 5)
```

- The keyword `return`

Example: Function with `return` statement

```
def bake_cookies(batch_size):
    num_cookies = batch_size * 12
    return num_cookies    # Returns the value of the variable num_cookies

# Call the function and store the result in the variable cookies_made
cookies_made = bake_cookies(2)
print(f"We baked {cookies_made} cookies!")
```

```
We baked 24 cookies!
```

- The `bake_cookies` function takes a parameter `batch_size` which shows how many batches of cookies to bake.
- It calculates the total number of cookies (`batch_size * 12`) and returns that value.
- You can then call the function and store the returned number of cookies in a variable.

Check your understanding

What code should be used in the highlighted section to return the value of `area`?

```
def calculate_area(base, height):
    area = 0.5 * base * height
    ???

triangle_area = calculate_area(3, 6)
```

Check your understanding

What code should be used in the highlighted section to return the value of `area`?

```
def calculate_area(base, height):
    area = 0.5 * base * height
    return area

triangle_area = calculate_area(3, 6)
```

Global and local variables

Global and local variables

In Python, variables are used to store data values.

A variable's **scope**, or where it can be accessed, is determined by where it is declared. You'll focus on two primary types of scope: global and local.

- A **global variable** is a variable that is assigned outside of a function definition and is available throughout the program after it's been assigned.
- A **local variable** is a variable that is assigned inside of a function definition and is available for use only within that function.

Global and local variables pt2

- While you can access global variables within functions, it's generally considered good practice to minimize their use. Try to use local variables whenever possible to keep your code well-organized. Overusing global variables can make your code harder to understand, debug, and maintain.
- For example, naming conflicts may occur if you accidentally use the same name for a global variable and a local variable.
- Additionally, global variables can cause unintended side effects if their values are modified unexpectedly in different parts of your code.

When to use what?

- Consider where the data needs to be accessed. If it is only within one function, you should use a local variable.
- If it needs to be accessed in multiple functions, you may want to use a global variable but this should be done with caution and an understanding that changing the value of a variable within a function will not change the value of the global variable.
- In general, it's best to minimize the use of global variables and use local variables and parameter passing for data sharing between functions.

Examine the code

- What global variable does the code block include?
- What local variable does the code block include?

```
def calculate_sum(a, b):
    result = a + b
    return result

sum = calculate_sum(5,10)

print(sum)
```

Examine the code

- What global variable does the code block include?
- What local variable does the code block include?

```
def calculate_sum(a, b):
    result = a + b
    return result

sum = calculate_sum(5,10)

print(sum)
```

- The code includes the global variable `sum`
- The code includes the local variable `result`

Example: Global and local variables

```
standard_oven_temp = 350  # Global variable (accessible everywhere)

def bake_cookies():
    flavor = "chocolate chip" # Local variable (specific to this function)
    print(f"Let's bake {flavor} cookies! Preheat the oven to
{standard_oven_temp}.")

def bake_cake():
    type = "birthday cake" # Local variable (specific to this function)
    print(f"Let's bake a {type} cake! Preheat the oven to
{standard_oven_temp}.")
```

Check your understanding

- What global variable(s) does the code block include?
- What local variable(s) does the code block include?

```
def celsius_to_fahrenheit(celsius):
    fahrenheit = (celsius * 9/5) + 32
    return fahrenheit

temp = celsius_to_fahrenheit(18)

print(temp)
```

Check your understanding

- What global variable(s) does the code block include?
- What local variable(s) does the code block include?

```
def celsius_to_fahrenheit(celsius):
    fahrenheit = (celsius * 9/5) + 32
    return fahrenheit

temp = celsius_to_fahrenheit(18)

print(temp)
```

- The code includes one global variable: `temp`
- The code includes two local variables: `celsius` and `fahrenheit`

IT'S CODING TIME

- Finding max value
- Average of three numbers
- Convert Fahrenheit to Kelvin

Questions?

Post to our Discord server

or

email:

upr-ciic3015-staff@upr.edu

karelys.lopez1@upr.edu

Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

CODE PATH .ORG

Modules

Where do functions come from?

`print()`, `int()`, `str()`

built-in functions: packaged with Python, always available for use.

Can also import additional functions

import statement: loads **module** which provides a set of functions to use

Use already-written functions!

material made by: Jean Mendez

Import Statement

```
print(math.ceil(3.4))
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

Import Statement

```
import math

print(math.ceil(3.4))
```

OR

```
from math import ceil

print(ceil(3.4))
```

material made by: Jean Mendez

Import Statement

All imports should be grouped together at the top of your file!

Bad

```
1  print('hi')
2
3  import math
4
5  print(math.ceil(1.8))
6
7  import random
8  num = random.randint(1, 100)
```

Good

```
1  import math
2  import random
3
4  print('hi')
5  print(math.ceil(1.8))
6  num = random.randint(1, 100)
```

material made by: Jean Mendez

import math

Access ALL mathematical utilities from math

```
math.ceil(3.14) == 4
math.factorial(4) == 24
math.sqrt(25) == 5
math.cos(2*math.pi) == 1.0
```

https://docs.python.org/3/library/math.html

Selective importing

```
from ... import ...
```

This syntax allows for selective importing.
Instead of importing the entire

randint()

randint(): function that takes two integers as inputs, returns a random integer between those two numbers, inclusive.

example: randint(0, 3) -> one of the integers 0, 1, 2, or 3 at random.

Included in **random** module

Example: flip a coin

```
from random import randint
```

randit: is a specific function within the **random module**.

random: is a built-in Python **module** that provides functions for generating pseudo-random numbers.

material made by: Jean Mendez

Example: flip a coin

```
from random import randint

number = randint(1,2)
if number == 1:
    print('Heads')
else:
    print('Tails')
```

Modules: Create your own!

- So far, we have been using built-in functions. These come from modules that are packaged with Python and are always available to use.
- However, a module is simply a file that contains Python code.
    - Thus, we can create our own modules and use them in other modules.

Modules: Create your own! Example

1. Create your own Python file and add a function to it!

Let's call this one *test_module.py*

```
# In test_module.py
def convert_celsius_to_fahrenheit(celsius):
    fahrenheit = 9 / 5 * celsius + 32
    return fahrenheit
```

material made by: Jean Mendez

Modules: Create your own! Example

2. Create a separate Python file in the same directory as ***test_module.py*** that will use it. Let's call this one ***file_that_uses_test_module.py***

```
# In file_that_uses_test_module.py

print("hello")
```

Modules: Create your own! Example

```
# In test_module.py

def convert_celsius_to_fahrenheit(celsius):

    fahrenheit = 9 / 5 * celsius + 32

    return fahrenheit
```

3. Import ***test_module.py*** into ***file_that_uses_test_module.py***.

```
# In file_that_uses_test_module.py

import test_module
```

material made by: Jean Mendez

Modules: Create your own! Example

```
# In test_module.py

def convert_celsius_to_fahrenheit(celsius):

    fahrenheit = 9 / 5 * celsius + 32

    return fahrenheit
```

4. Now that ***test_module.py*** is imported, you can call any functions declared in it as follows:

```
# In file_that_uses_test_module.py

import test_module

print(test_module.convert_celsius_to_fahrenheit(40))
```

material made by: Jean Mendez

IT'S CODING TIME

- Let's create a temperature module and call it!

Questions?

Post to our Discord server

or

email:

upr-ciic3015-staff@upr.edu

karelys.lopez1@upr.edu

Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

CODE PATH .ORG